Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 167-184

COMPARING AND EVALUATING DESIGN PATTERN MINER TOOLS

L. J. Fülöp, Á. Ilia, Á. Z. Végh, P. Hegedűs and R. Ferenc

(Szeged, Hungary)

Abstract. Several tools are published in the literature which are able to mine design pattern usage from source code. Because a common test database – a benchmark – is not available, the accuracy of the tools is difficult to check and measuring any kind of improvements on the tools is also problematic. As an all-in-one solution we have developed a benchmark for evaluating and comparing design pattern miner tools and for ensuring a test database for them.

In this paper we present some experiments performed with the benchmark. Two design pattern miner tools – Columbus and Maisa – are evaluated and compared. The tools are evaluated on C++ reference implementations of design patterns, on a real software system called NotePad++ and on FormulaManager, which is a software implemented by us to have a test case where the usage of design patterns is well defined and documented. Design pattern instances from NotePad++ recovered by professional software developers are also added to the benchmark.

1. Introduction

The development process of software systems contains several steps. Due to close deadlines most of the projects skip partly or completely less important parts of these steps. Making up-to-date documentation is usually the least important task for programmers (at least in their opinion), so it is partly or completely skipped which causes a lot of problems later in maintenance and evolution phases

of the software. The problem of missing documentation can be moderated by design pattern mining, because design patterns [7] play important role in a software, so knowing about them helps program comprehension. For this reason several design pattern mining tools are developed and published.

The problem with several approaches to pattern recognition (based on pattern matching) is that they are inherently too permissive in a sense that *they produce many false results* in which some code fragments are identified as pattern instances that share only the structure of the pattern description. The pattern miner tools usually use different definitions of patterns and algorithms to detect these patterns. Another major problem is that *there is no common measurement system or a test database* to evaluate and compare results produced by design pattern miner tools. Furthermore, we experienced demand in conferences and publications for a solution to evaluate patterns effectively and easily [15]. Hence, *we developed a publicly available benchmark* for evaluating and comparing design pattern miner tools. The developed benchmark is general, it is language, software, tool and pattern independent. With the benchmark the *accuracy (precision and recall) of the tools can be validated by the public.*

In this paper the benchmark will be introduced by experiments on evaluating and comparing two design pattern miner tools, *Maisa* and *Columbus*. The tools are evaluated on reference implementations of design patterns, on a software system called FormulaManager, which contains every design pattern in a real context, and on an open source software system called NotePad++. In addition, further instances from NotePad++ recovered by professional software developers are added to the benchmark.

We will proceed as follows. In the next section we will discuss some works similar to ours. The benchmark will be described shortly in Section 3. Afterwards, the two evaluated tools (Columbus and Maisa) will be introduced in Section 4. We will show the evaluation and comparison results of the tools in Section 5. Finally, in Section 6 we will present some conclusions and outline directions for future work.

2. Related work

In our previous work we have presented a method to differentiate true and false hits in design pattern mining [4]. We employed machine learning methods to filter out false design pattern hits. First, we ran our design pattern miner tool that discovers patterns based on structural descriptions. Afterwards, we classified these hits as being true or false manually, and finally we calculated predictive information for the hits. We trained a decision tree based on classified

values and on the predictive information, from which we were able to mine true design pattern hits more accurately. With the presented benchmark it will be faster and easier to classify the results of a pattern miner tool.

Recently, we presented a comparison of three pattern miner tools [6]: Columbus [5], Maisa [14] and CrocoPat [2]. We compared them regarding patterns hits, speed and memory consumption. We gave the same common input to the tools created by the source code analyzer front end and the exporter plug-ins of the Columbus framework. We concluded that CrocoPat was the fastest, Maisa required the least memory, while Columbus was an all-in-one solution for pattern detection from C++ source code with comparable performance to the other two specialized tools. In this work we did not examine the accuracy of the found design pattern instances. We have continued this work by developing the benchmark and performing experiments on it.

Petterson et al. [15] summarized occurring problems during the evaluation of accuracy in pattern detection. The goal was to make accuracy measurements more comparable. Six major problems were revealed: design patterns and variants, pattern instance type, exact and partial match, system size, precision and recall, and control set. A control set was "the set of correct pattern instances for a program system and design pattern." The determination of the control sets was very difficult, therefore solutions from natural language parsers were considered. One good solution was the *tree banks*. Tree banks could be adapted by establishing a large, manually validated pattern instances database. Another adaptable solution was the idea of pooling process: "The idea is that every system participating in the evaluation contributes a list of n top ranked documents, and that all documents appearing on one of these lists are submitted to manual relevance judgement." The process of constructing control sets had two main problems. First, they were not complete in most software systems. Second, on a real scale software system a single group was not able to determine a complete control set. The authors stated that *community effort is highly required* to make control sets for a set of applications.

Guéhéneuc et al. [8] introduced a comparative framework for design recovery tools. The purpose of the authors' framework was not to rank the tools (Ptidej and LiCoR) but to compare them on the basis of their qualitative aspects. This framework contained eight aspects: context, intent, users, input, technique, output, implementation and tool. There is a major need for this framework since the comparison between design recovery tools is very difficult due to the fact that they have very different characteristics in terms of representation, output format and implementation techniques. This framework provides an opportunity for comparing not only similar systems, but also systems which are different. Our work is different from this one because we use a *benchmark* and *tool* to evaluate design pattern instances and to compare design pattern miner tools based on their results.

3. Benchmark

We use the well-known issue and bug tracking system called Trac [17] (version 0.9.6) as the basis of the benchmark. Trac is written in Python and it is an easily extendible and customizable plug-in oriented system. The aim of Trac is to give an easy and efficient way for tracking bugs and issues. Issue tracking is based on *tickets*, where a ticket stores all information about an issue or a bug. A ticket is identified by a unique number.

Although the Trac system provides many useful services, we had to do a lot of customization and extension work to create a benchmark from it. The two major extensions were the customization of the graphical user interface and the customization of the system's tickets. In the case of the tickets we had to extend them to be able to describe design pattern instances (name of the pattern, information about its participants, information about its evaluation, etc). In the case of the graphical user interface we had to inherit and implement some core classes of the Trac system. For now, we give only a short overview of the benchmark's functionalities, the customized graphical interface will be presented in detail during the evaluation and comparison of Columbus and Maisa in Section 5. A detailed description of the usage of the user interface can be found in the wiki pages of the benchmark (*http://www.inf.u-szeged.hu/designpatterns/*).

The benchmark contains three main menu points: *evaluation*, *upload* and *register*. From the evaluation menu point three important views can be accessed. These are the *statistics view*, *comparison view* and the *instance view*. In the instance view the pattern instances can be categorized by two aspects, correctness and completeness. *Completeness* means how complete the evaluated pattern instance is in a structural sense. More precisely, it means how many pattern participants can be found in the instance. *Correctness* means how correct the evaluated pattern instance is in a behavioural sense. More precisely, it means to what degree the pattern instance matches the original intent of the design pattern. Registration is required to evaluate pattern instances. From the upload menu point the *new language* functionality, the *new software* functionality, the *new tool* functionality and the *new instances* functionality can be accessed. Instances recovered by people can be uploaded as well, in this case the name of the tool is "Human".

The benchmark calculates two well-known and important accuracy measurements: *precision* and *recall*. Explaining the meaning of precision and recall requires the following definitions.

- *True Positives* (TP): true instances found by the tool (correctly).
- *False Positives* (FP): false instances found by the tool (incorrectly).
- *False Negatives* (FN): true instances not found by the tool (incorrectly).

The *precision* value is defined as $\frac{TP}{TP+FP}$, which means the ratio of correctly identified instances with respect to all found instances. The *recall* value is defined as $\frac{TP}{TP+FN}$, which means the ratio of correctly identified instances with respect to all existing real instances.

The benchmark contains 1,274 design pattern instances from three C++ software systems (Mozilla [12], NotePad++ [13] and FormulaManager [16]), three Java software systems (JHotDraw [9], JRefactory [10] and JUnit [11]) and C++ reference implementations of design patterns. The uploaded design pattern instances are recovered by three design pattern miner tools: Columbus (C++), Maisa (C++) and Design Pattern Detection Tool (Java) [18], [3]. In this paper Columbus and Maisa are evaluated and compared by using the benchmark.

Siblings and fundamental participants. Several design patterns are structured in a way which allows e.g. multiple concrete implementations of an abstract class. Some tools are able to recognize this situation, but others report multiple pattern instance hits in these cases (one for each concrete implementation). The benchmark is able to subsequently assign these instances, which are referred to as *siblings*, to each other.

The determination of siblings is based on the *fundamental participants* of design patterns. A fundamental participant occurs only once in a design pattern instance. For example, in the case of the State pattern the fundamental participant is the State class, while Context and ConcreteState classes can be repeated.

4. Evaluated Tools

4.1. Maisa

Maisa is a software tool [14] for the analysis of software architectures developed in a research project at the University of Helsinki. The key idea in Maisa is to analyze design level UML diagrams and compute architectural metrics for early quality prediction of a software system. In addition to calculating traditional (object-oriented) software metrics such as the Number of Public Methods, *Maisa looks for instances of design patterns* (either generic ones such as the well-known GoF patterns or user-defined special ones) in UML diagrams representing the software architecture.

UML diagrams are represented with a description language very similar to Prolog. The description language is used to represent the source code and design patterns as well. Thus, Maisa can match design patterns by using the description language.

4.2. Columbus

Columbus is a reverse engineering framework, which has been developed in cooperation between FrontEndART Ltd., the University of Szeged and the Software Technology Laboratory of Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract facts from them. The main motivation to develop the Columbus system has been to create a general framework to combine a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are provided as basic parts of Columbus, while the system can be extended to meet other reverse engineering requirements as well. This way we have got a versatile and easily extendible tool for reverse engineering.

In this work Columbus provides the common framework. Figure 1 shows

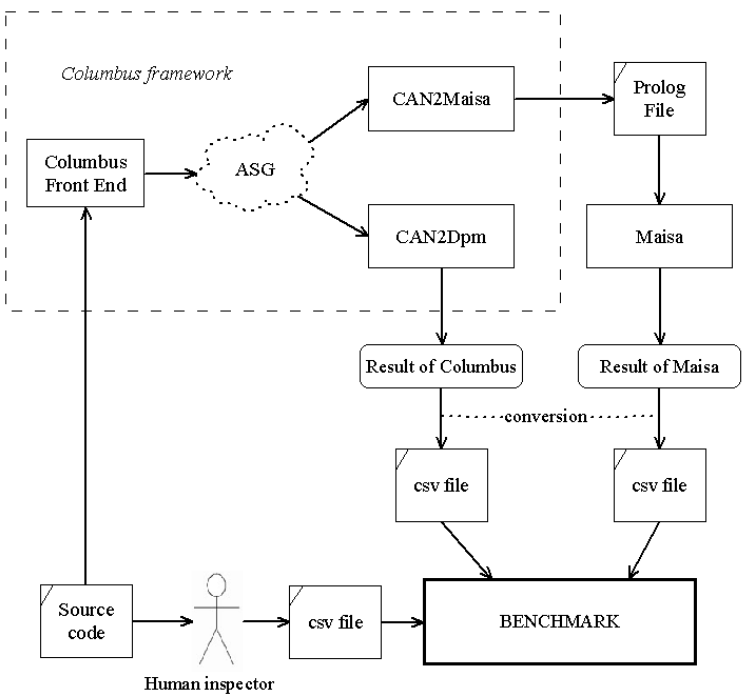


Figure 1. Framework for design pattern mining

the specialization of the framework for design pattern mining. First Columbus analyzes the source code and builds an ASG. The CAN2Maisa component of Columbus creates a UML class diagram from the ASG and saves this information in the required format for Maisa. After that, Maisa discovers pattern instances from this file. The CAN2Dpm component of Columbus mines pattern instances directly from the ASG. The design patterns are described in DPML (Design Pattern Markup Language) files, which store information about the structures of the design patterns. CAN2Dpm tries to match this graph to the ASG using our algorithm described in previous work [1]. Finally, the results of the tools are uploaded into the benchmark after they are converted to the required CSV file format presented in the following.

Upload file format. The format of the CSV file containing the design pattern instances must be the following (see Figure 2). Each pattern instance consists of several lines closed by an empty line. The first line of every instance is always the name of the pattern, without any white space characters. If the name consists of several words, then they have to be concatenated and every new word should start with an upper case letter. The subsequent lines represent the participants of the instance. These lines contain values separated by commas. The first value contains the role and the original name of the pattern participant. The role can be *class*, *operation* and *attribute*. If the participant is fundamental (see Section 3), then a star has to be given before the role in the first value. The second value contains the real name of the participant in the source code. Finally, the third value contains the relative path of the participant in the source code together with starting and ending line information separated by colons.

```
TemplateMethod
*class AbstractClass,MyAbstractClass,TemplateMethod.h:1:6
operation TemplateMethod,MyTemplateMethod,TemplateMethod.cpp:3:15
operation PrimitiveOperation,MyPrimitiveOperation1,TemplateMethod.h:4:4
class ConcreteClass,MyConcreteClass,TemplateMethod.h:8:12
operation PrimitiveOperation,MyPrimitiveOperation1,TemplateMethod.h:9:9
```

Figure 2. Example CSV file

The uploaded pattern instances are checked if they already exist in the database. If a known pattern instance is found again by the new tool and the two instances are exactly the same, then the tool entry of the instance is extended by the name of the new tool and no new instance will be created in the database. If only the fundamental participants(see Section 3) are the same, then a new instance will be created and the instances will be connected as siblings.

5. Evaluation of Columbus and Maisa

In this section Columbus and Maisa are evaluated and compared by using the benchmark. The tools are evaluated on C++ reference implementations of design patterns, on NotePad++, and on a software called FormulaManager implemented by us to have a test case where the usage of design patterns is well defined and documented. The reference implementations test the structural matching ability of the tools while in case of NotePad++ and FormulaManager the tools are examined in a real context where other factors are also considered, e.g. the aim of the pattern.

Reference implementations. To compare different tools, reference implementations of design patterns based on work of Gamma et al. [7] were created. With these reference implementations the basic capabilities of C++ pattern miner tools can be evaluated and compared.

Design pattern mining strongly depends on analyzing the source code. Therefore, the source files in the reference implementations do not contain any difficult programming structures like templates and they do not include any standard headers to avoid parsing problems. Furthermore, a common framework is provided by Columbus as we presented it in Section 4, so the inputs of the tools are the same.

Let us take a concrete reference implementation, the *Adapter Object*. First we give a short description of this pattern. The aim of the Adapter pattern is to "convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces" [7]. The Adapter pattern has four participants (see Figure 3). First, the *Target* class defines the domain-specific interface that the Client uses. *Client* in turn represents the class collaborating with objects that conform to the Target interface. Next, the *Adaptee* class describes an existing interface that needs adapting, and finally *Adapter* is the class that adapts the interface of

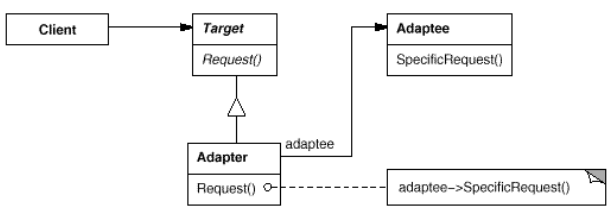


Figure 3. The Adapter Object design pattern

Adaptee to the Target interface. To adapt one interface to another the Adapter Object pattern uses composition. The reference implementation of Adapter Object is shown in Figure 4. In reference implementations every participant class starts with the "Example_" prefix, similarly attributes start with the "example_" prefix.

```
class Example_Target {
    public:
        virtual void function() = 0;
};

class Example_Adaptee {
    public:
        virtual void other_function();
};

class Example_AdapterObject : public Example_Target {
    public:
        virtual void function();
    private:
        Example_Adaptee* m_adaptee;
};

void Example_Adaptee::other_function() { }

void Example_AdapterObject::function() {
    m_adaptee->other_function();
}
```

Figure 4. Reference implementation of Adapter Object

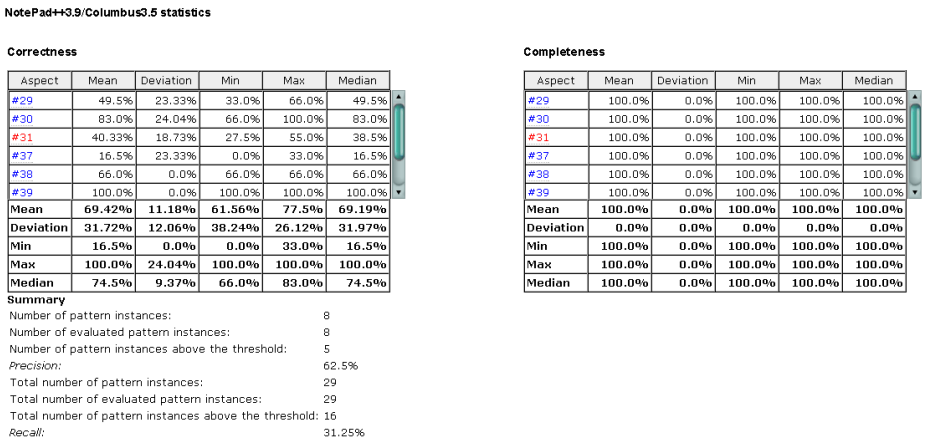
NotePad++ is a free source code editor which supports several programming languages, running in Microsoft Windows environments. It is based on the Scintilla edit component (a very powerful editor component), written in C++ with pure win32 API and STL. Calculated by the metric component of Columbus, NotePad++ contains 95 classes, 1,371 methods, 776 attributes and 27,033 useful lines of code.

When evaluating design pattern miner tools on real software systems, it is necessary to consider not only instances mined by the tools but instances recognized by programmers, too. If instances mined by programmers are skipped from an evaluation then the recall of the tools cannot be calculated. Therefore instances from NotePad++ recovered by programmers were also added to the benchmark and were considered during the evaluation of Columbus and Maisa.

FormulaManager. Since the reference implementations contain disjoint implementations of the design patterns in an unreal context, we developed a software called FormulaManager where every design pattern occurs in a real context at least once. The aim of FormulaManager is to manage formulas. Some functionalities are: executing operations on formulas (evaluation, prefix form, postfix form), exporting the results to different kinds of representations (HTML, XML,

CSV), displaying the results in different views (list, table), handling different numbering systems, validating a formula, etc.

Table 1 presents the results of the tools, where the upper part shows the number of instances found. In the lower part of the Table a summary is shown as well, where precision, recall, correctness mean and completeness mean are provided by the benchmark's statistics view (see Figure 5). A dash in Table 1 represents that the tool does not search for the given pattern instance.



NotePad++3.9/Columbus3.5 statistics

Correctness

Aspect	Mean	Deviation	Min	Max	Median
#29	49.5%	23.33%	33.0%	66.0%	49.5%
#30	83.0%	24.04%	66.0%	100.0%	83.0%
#31	40.33%	18.73%	27.5%	55.0%	38.5%
#37	16.5%	23.33%	0.0%	33.0%	16.5%
#38	66.0%	0.0%	66.0%	66.0%	66.0%
#39	100.0%	0.0%	100.0%	100.0%	100.0%
Mean	**69.42%**	**11.18%**	**61.56%**	**77.5%**	**69.19%**
Deviation	**31.72%**	**12.06%**	**38.24%**	**26.12%**	**31.97%**
Min	**16.5%**	**0.0%**	**0.0%**	**33.0%**	**16.5%**
Max	**100.0%**	**24.04%**	**100.0%**	**100.0%**	**100.0%**
Median	**74.5%**	**9.37%**	**66.0%**	**83.0%**	**74.5%**

Completeness

Aspect	Mean	Deviation	Min	Max	Median
#29	100.0%	0.0%	100.0%	100.0%	100.0%
#30	100.0%	0.0%	100.0%	100.0%	100.0%
#31	100.0%	0.0%	100.0%	100.0%	100.0%
#37	100.0%	0.0%	100.0%	100.0%	100.0%
#38	100.0%	0.0%	100.0%	100.0%	100.0%
#39	100.0%	0.0%	100.0%	100.0%	100.0%
Mean	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Deviation	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**
Min	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Max	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**
Median	**100.0%**	**0.0%**	**100.0%**	**100.0%**	**100.0%**

Summary

Number of pattern instances:	8
Number of evaluated pattern instances:	8
Number of pattern instances above the threshold:	5
Precision:	62.5%
Total number of pattern instances:	29
Total number of evaluated pattern instances:	29
Total number of pattern instances above the threshold:	16
Recall:	31.25%

Figure 5. Columbus statistics on NotePad++

The statistics view contains two tables, one for *correctness* and one for *completeness*, respectively. Every line in the upper part of the tables means a statistic for one particular pattern instance. A pattern instance can be evaluated by numerous people, therefore mean, deviation, minimum, maximum and median values can be calculated for the instance. The lower part of the tables contains the same kind of basic statistics but they are calculated from the statistics of the instances (shown in the upper part of the tables).

The benchmark can be customized to consider sibling relations or not when making statistics. The default option is to handle sibling instances as only one common instance. For example in the case of Reference Implementations Columbus discovers two Abstract Factory instances, while Human discovers only one. Difference is caused because Human discovers sibling instances as one pattern instance, while Columbus discovers them as two instances.

Software	Reference Implementations			NotePad++			FormulaManager		
Tool	Columbus	Maisa	Human	Columbus	Maisa	Human	Columbus	Maisa	Human
Abstract Factory	2	2	1	0	0	0	4	0	2
Adapter Class	1	1	1	0	4	0	2	1	1
Adapter Object	1	1	1	9	4	0	4	0	1
Bridge	1	-	1	0	-	0	6	-	2
Builder	1	1	1	0	1	0	1	0	1
Chain Of Resp.	0	-	1	0	-	0	1	-	1
Command	-	-	1	-	-	0	-	-	1
Composite	-	-	1	-	-	1	-	-	1
Decorator	1	-	1	0	-	0	1	-	1
Facade	-	-	1	-	-	0	-	-	1
Factory Method	1	0	1	0	0	0	2	3	4
Flyweight	-	0	1	-	0	0	-	1	1
Interpreter	-	-	1	-	-	1	-	-	1
Iterator	1	0	1	0	0	1	1	0	1
Mediator	-	0	1	-	0	0	-	0	1
Memento	-	1	1	-	0	0	-	0	1
Observer	-	0	1	-	0	1	-	0	1
Prototype	1	1	1	0	6	1	3	2	2
Proxy	1	1	1	0	11	1	2	0	1
Singleton	0	0	1	0	0	2	3	0	3
State	1	-	1	1	-	0	9	-	1
Strategy	1	-	1	0	-	0	2	-	2
Template Method	1	-	1	3	-	1	1	-	1
Visitor	1	4	1	0	0	0	2	3	1
Precision	100%	80.00%	100%	62.50%	16.67%	90.91%	52.27%	80%	100%
Recall	58.33%	33.33%	100%	29.41%	11.76%	58.82%	71.88%	25%	100%
Correctness mean	100%	80.00%	100%	69.42%	20.63%	77.05%	55.23%	85.95%	100%
Completeness mean	96.43%	85.00%	100%	100%	92.36%	90.91%	85.98%	48.02%	100%

Table 1. Number of found design pattern instances with considering sibling relations

5.1. Reference implementations

In this section we summarize our experiments on evaluating the tools on reference implementations. The number of pattern instances found by the tools in reference implementations are listed in Table 1. Pattern instances found by programmers (column called *Human*) in the reference implementations are real true pattern instances.

Some design patterns are more difficult to discover than others. This is due to their unclear structure even if the aim of such a design pattern is obvious. The purpose of the unclear specification is to allow flexible implementations of the design pattern. For example, in case of Mediator, only an abstract incomplete structural specification is given, e.g. no methods are specified in the participant classes. Most of the tools consider only structural information of the patterns, therefore they encounter problems in this case. A promising solution is presented by Wendehals [19] to improve design pattern discovering by using dynamic information.

Because the two evaluated tools use only structural information, they do not discover most of the patterns with unclear specification, e.g. Command, Facade and Interpreter as it can be seen in Table 1. Although Maisa tries to discover some patterns that have unclear specification like Mediator, Memento, Observer and Flyweight, it works only in case of Memento (instance #1152). Because Maisa misses these patterns its recall value is only 33.33%.

If more than one pattern instance was discovered in the reference implementations for a given pattern than either *false positives were found* or *the pattern instance was recognized several times*. For example Maisa discovers four Visitor pattern instances. We examined these instances in the benchmark and realized that two of them were false positives, while the other two instances were actually the same instance with common abstract but with different concrete participants. The two true instances are connected as siblings (see Section 3) to each other and to the single pattern instance found by Columbus and Human. This way these instances appear as one common instance in the benchmark.

Figure 6 shows the instance view of one of the two false Visitor instances mined by Maisa (instance #1169). First, let us see what can be found on the instance view. The participants of the actual pattern instance are shown in the top left corner with their concrete name in the source code. When the user clicks on a participant's name, the source code is shown with the pattern instance highlighted in the right hand side of the view. The two categorization queries, correctness and completeness (see Section 3), are located below the participants. The categorization can be applied to the *siblings* of the pattern instance as well (if there are any) by selecting the corresponding checkboxes. The source of Example_ConcreteElement2 class is loaded into the right hand side of the view because it was selected previously from the participants. It is a false instance

Figure 6. Visitor instance mined by Maisa from reference implementations

because Example_ConcreteElement2 acts as a class Element participant as it can be seen in the area of the participants (see Figure 6). Maisa discovers another false instance with the same problem (#1171).

In case of Abstract Factory both Maisa and Columbus discover two pattern instances. These pattern instances are true, the only problem is that neither tool can group concrete elements of the Abstract Factory pattern, therefore they discover the same pattern instance twice with different concrete participants. It causes the mean of Columbus' completeness to be 96.43%. Maisa encounters a similar problem in case of the Visitor pattern, therefore its mean of completeness is 85%.

Comparison. The instances found by Columbus, Maisa and humans are compared by the comparison view functionality of the benchmark (see Figure 7). The comparison lists pattern instance numbers in different views. By clicking on the numbers the concrete design pattern instance can be examined in the instance view.

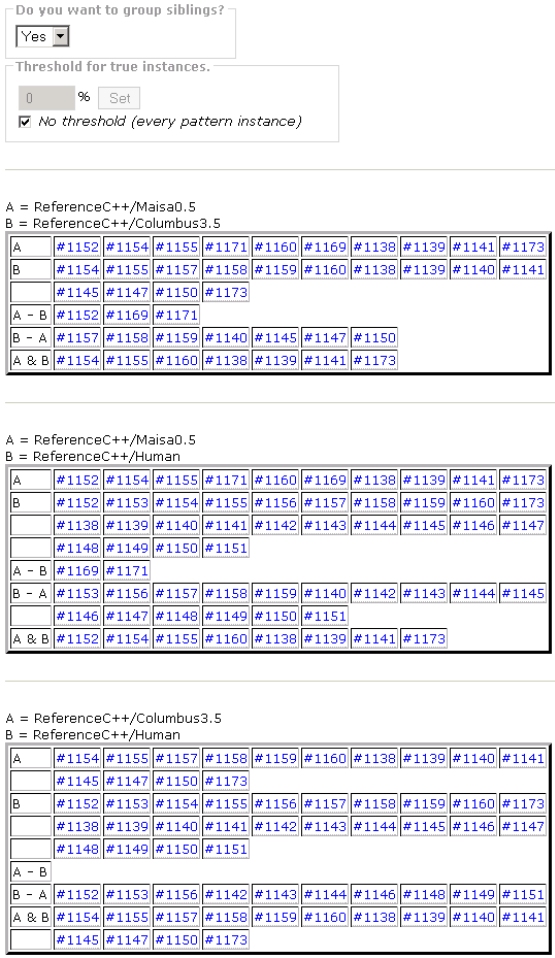


Figure 7. Comparison of instances in reference implementations

Let us see the first table in Figure 7 as an example. This table shows the comparison of Maisa and Columbus. The difference of Maisa and Columbus (row A-B) contains three pattern instances. The previously mentioned false positive Visitor instances (#1169 and #1171) are found by Maisa only, therefore these appear in the difference. The other difference is a true pattern instance of Memento (#1152), that is found by Maisa and not found by Columbus. Because Columbus finds several design patterns that Maisa does not a lot of instances appear in the difference of Columbus and Maisa (row B-A). But the tools find some patterns in common with the same results that appear in the intersection (row A&B).

A correctness threshold can be set to determine which is a true and which is a false pattern instance based on human evaluation. The threshold can be set in the upper part of the comparison view (see Figure 7). After a threshold is set only the instances with higher mean of correctness are considered in the comparison. For example if the threshold is set to 50% in the previous comparison, the two false positive Visitor pattern instances mined by Maisa will not appear.

The comparison view contains two further tables containing the comparison of Maisa and Columbus with the results of human inspection of the code, respectively. The instances found by careful human inspection represent the desirable results of a tool. Therefore, the difference between e.g. Human and Columbus represents the *false negatives* for Columbus. This value is used for calculating the recall values.

5.2. NotePad++

After examining the tools on the reference implementations we performed another test, but this time on a real project, NotePad++. The results of the tools can be seen in the middle three columns of Table 1 which will be explained in the following.

Columbus found only a few types of patterns (Adapter Object, State and Template Method), however with better precision than the ones (Adapter Class, Adapter Object, Builder, Prototype and Proxy) found by Maisa. It can also be noticed that software developers tend to find those pattern instances, which are difficult to find by pattern miner tools because of their unclear specifications. Therefore, we assume that instances found by humans are a good amendment of automatic tools even if software developers will probably not find all the pattern instances in a particular project. To prove this assumption we want to analyze further software systems manually in the future.

In Table 1 it can be noticed that the precision of Human on NotePad++ is not 100%. It is due to the fact that the result of Human in the benchmark consists of several people which may disagree whether a pattern instance fulfills the original intent of the design pattern. This is one of the great benefits of the benchmark that it lets people vote whether the uploaded pattern instances are real patterns or not. Even though human analysis resulted in the highest recall (58.82%) it is still far from 100% which indicates that the current amount of human analysis work put into the benchmark is not enough to find all the pattern instances in larger projects.

In Figure 5 the correctness and completeness tables show that even structurally completely correct pattern instances are sometimes voted to be false instances (their structure is adequate, but it is only a coincidence). Again it is due to the fact that evaluated tools rely solely on the structure of the pattern.

At the bottom of Figure 5 the precision (62.5%) and recall (31.25%) values are also shown. Although Maisa found more types of pattern than Columbus, its precision (16.67%) and recall (11.76%) are lower than those of Columbus. It can also be seen on Figure 5 that the average deviation of correctness is relatively high (11.18%) which can be explained by the fact that certain pattern types are described better by their structure while others need behavioral analysis to be discovered. Similar statistics can be queried for the other tools on NotePad++ and on reference implementations as well. The collected results in Table 1 are based on these statistics views.

5.3. FormulaManager

In this section, we summarize our experiments on evaluating the tools on FormulaManager. We note that FormulaManager differs from the simple reference implementations because the design patterns are used in a real context and they are connected to each other. The numbers of the pattern instances found by the tools in FormulaManager are listed in the last three columns of Table 1.

Let us examine and compare the tools, Columbus and Maisa, on FormulaManager. It is clear that Maisa has a good precision (80%) but its recall is poor (25%). The poor recall is due to the fact that Maisa misses the instances of several design patterns (e.g. Builder, Mediator, Proxy) and it does not try to discover every kind of pattern (e.g. Command, Decorator). On the contrary, Columbus has a fair recall (71.88%) and precision (52.27%). Columbus discovers several false positives in the case of State, Bridge and Adapter Object, which results in its lower precision compared to Maisa. In case of the other patterns Columbus has good precisions.

6. Conclusion

In this paper we presented experiments performed on a newly developed benchmark for evaluating and comparing design pattern miner tools. The benchmark is general considering the mined software systems, programming languages, uploaded design pattern instances and design pattern miner tools.

With the help of the benchmark the accuracy of two design pattern miner tools (Columbus and Maisa) were evaluated on reference implementations of design patterns and on two software systems, NotePad++ and FormulaManager. In this work design pattern instances used in NotePad++ are discovered also manually, so both precision and recall values could be calculated by the benchmark. Furthermore, we developed a software system called FormulaManager to

test the tools on a software where every design pattern is implemented in a real context.

Sometimes false instances were found by the tools because they are examining only the structure of the code while programmers also take the code's behavior into consideration. That is why instances mined by humans can be used to provide an important extension to results of the tools. In the future we want to involve other research groups to discover large software systems by hand, such as Mozilla and Eclipse.

With the help of the benchmark it is possible to evaluate design pattern miner tools which will hopefully lead to better quality tools in the future. The benchmark can be *freely accessed* from the home page of the Institute of Informatics at the University of Szeged:

http://www.inf.u-szeged.hu/designpatterns/

References

[1] **Balanyi Zs. and Ferenc R.,** Mining design patterns from C++ source code, *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, IEEE Computer Society, 2003, 305–314.

[2] **Beyer D. and Lewerentz C.,** CrocoPat: Efficient pattern analysis in object-oriented programs, *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, IEEE Computer Society, 2003, 294–295.

[3] *The Design Pattern Detection tool Homepage*, `http://java.uom.gr/~nikos/pattern-detection.html`.

[4] **Ferenc R., Beszédes Á, Fülöp L. and Lele J.,** Design pattern mining enhanced by machine learning, *Proceedings of the 21th International Conference on Software Maintenance (ICSM 2005)*, IEEE Computer Society, 2005, 295–304.

[5] **Ferenc R., Beszédes Á., Tarkiainen M. and Gyimóthy T.,** Columbus – Reverse engineering tool and schema for C++, *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society, 2002, 172–181.

[6] **Fülöp L., Gyovai T., and Ferenc R.,** Evaluating C++ design pattern miner tools, *Proceedings of the 6th International Workshop on Source Code*

Analysis and Manipulation (SCAM 2006), IEEE Computer Society, 2006, 127–136.

[7] **Gamma E., Helm R., Johnson R. and Vlissides J.,** *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley Publ. Co., 1995.

[8] **Guéhéneuc Y.-G., Mens K. and Wuyts R.,** A comparative framework for design recovery tools, *Proceedings of the 10th Conference on Software Maintenance and Reengineering(CSMR'06)*, IEEE Computer Society, 2006, 123–134.

[9] *The JHotDraw Homepage*, `http:/www.jhotdraw.org`.

[10] *The JRefactory Homepage*, `http:/jrefactory.sourceforge.net/`.

[11] *The JUnit Homepage*, `http:/www.junit.org`.

[12] *The Mozilla Homepage*, `http://www.mozilla.org`.

[13] *The NotePad++ Homepage*, `http://notepad-plus.sourceforge.net/`.

[14] **Paakki J., Karhinen A., Gustafsson J., Nenonen L. and Verkamo A.I.,** Software metrics by architectural pattern mining, *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, 2000, 325–332.

[15] **Pettersson N., Löwe W. and Nivre J.,** On evaluation of accuracy in pattern detection, *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, 2006.

[16] *The source code of FormulaManager*, `http://www.sed.hu/src/FormulaManager/`.

[17] *The Trac Homepage*, `http://trac.edgewall.org/`.

[18] **Tsantalis N., Chatzigeorgiou A., Stephanides G. and Halkidis S.T.,** Design pattern detection using similarity scoring, *IEEE Transactions on Software Engineering*, **32** (2006), 896–909.

[19] **Wendehals L.,** Improving design pattern instance recognition by dynamic analysis, *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, May 2003*, 29-32.

L.J. Fülöp, Á. Ilia, Á.Z. Végh, P. Hegedűs and R. Ferenc

Department of Software Engineering
University of Szeged
Árpad tér 2.
H-6720 Szeged, Hungary
{flajos, ferenc}@inf.u-szeged.hu
ilia.arpad@stud.u-szeged.hu
{vegh.adam.zoltan, hegedus.peter.3}@stud.u-szeged.hu